20. Алгоритми върху едномерни масиви

Проф. д-р Емил Хаджиколев

1. Основни алгоритми за обработка на масиви.
   - Сума на елементите на масив;
   - Средно-аритметично на елементите на масив;
   - Търсене на елемент по зададена стойност;
   - Намиране на минимален и максимален елемент в масив.
2. Алгоритми за сортиране
   - Сортиране чрез вмъкване;
   - Сортиране чрез пряка селекция;
   - Метод на мехурчето;
   - Бързо сортиране;
3. Двоично търсене.

Основни алгоритми за обработка на масиви

- намиране на сума и средно-аритметично;
- намиране на минимален и максимален елемент;
- търсене на елемент по стойност;
- сортиране;
- двоично търсене (в сортиран масив) и др.

- Повечето алгоритми се използват и при работа с не-числови масиви.
- Алгоритмите са описани във функции, получаващи масива като параметър.
- За коректната работа на функциите трябва да се отчете фактът, че променливата за масив може да има стойност NULL и да се извърши подходяща проверка. Това в разглежданите примери не е направено, за да не се усложнява основната логиката.

Сума на елементите на масив

- За да се намери сумата на елементите на масив, **масивът може се обходи в цикъл и при всяка итерация текущия елемент да се добавя в променлива за сума**.
- Използва се функцията (от лекция 15), която връща масива във вид на низ.

```
string arrayToString(int arr[], int size) {
    string s = "{";

    for (int i = 0; i < size; i++) {
        s += to_string(arr[i]) + (i<size - 1 ? ", " : "");
    }
    s += "}";

    return s;
}
```

Сума на елементите на масив - решение

```
int sum(int arr[], int size) {
    int sum = 0;                          // в началото сумата е 0
    for (int i = 0; i < size; i++) {      // при всяка итерация
        sum += arr[i];                    // към сумата добавяме текущия елемент
    }
    return sum;                           // връща се сумата
}
int main() {
    setlocale(LC_ALL, "bg");
    const int size = 7;
    int arr[size] = {5, 6, -3, 8, -2, 1, 5 };

    string asString = arrayToString(arr, size);
    int arrSum = sum(arr, size);

    cout << "Масив: " << asString << '\n';
    cout << "сума: " << arrSum << '\n';

    return 0;
}
```

Средно-аритметично на елементите на масив

- Средно-аритметична стойност се получава, като се **раздели сумата от елементите на масива на техния брой**.
- Средно-аритметичната стойност може да не е цяло число. **За да не се получи цяло число при делението на две цели (сума/брой)**, **едното цяло число се преобразува до реално** и чак след това се извършва делението.
- След като логиката за сума е вече реализирана може да се използва създадената за целта функция.

Средно-аритметично на елементите на масив - решение

```
double average(int arr[], int size) {
      return (double)sum(arr, size) / size;
}

int main() {
   ...
   cout << "ср.ар.: " << average(arr, size) << '\n';

   return 0;
}
```

Търсене на елемент по зададена стойност (1)

- **Задачата при търсенето е да се провери дали дадена стойност се намира в масив**.
- **Функцията може да връща** булева стойност (true, false) или индекс на масива, в зависимост от това, как точно е формулирана задачата.
- **Повече информация носи функция, връщаща индекса на намерения елемент**. Ако не е намерен елемент, функцията може да връща, например числото -1, което не е валиден индекс.
- **Ако е необходимо функцията да връща индекс**, трябва да се отчете фактът, че в масива може да има няколко елемента с търсената стойност. Тогава функцията може да връща индекса на първия намерен елемент, на последния или списък (напр. масив) с всички намерени индекси.

Търсене на елемент по зададена стойност (2)

- **Тук е реализирана задачата за извеждане на последния намерен индекс.**
- Показано е **обхождане на масив отзад напред**.
- **Ако масивът е неподреден**, в процеса на търсене последователно се обхождат всички елементи (в случая отзад напред). Текущата стойност на масива се сравнява с търсената стойност и при съвпадение, функцията приключва, като връща намерения индекс.
- **Ако масивът е подреден**, има по-бърз алгоритъм за двоично търсене.

Търсене на елемент по зададена стойност - решение

```
int search(int arr[], int size, int searchVal) {
    // Обхождаме масива, например, отзад напред
    for (int i = size-1; i>=0; i--){
        if(arr[i]==searchVal){   // ако елемента е намерен,
               return i;         // връщаме позицията му
        }
    }
    return -1;                   // връщаме -1, ако не е намерен
}

int main() {
    ...
    int searchVal = 6;                                  // търсена стойност
    int searchPos = search(arr, size, searchVal); // намерена позиция
    cout << "Елемент " << searchVal <<
    (searchPos == -1?" не е намерен":" е намерен на позиция " + to_string(searchPos))
     << '\n';

    return 0;
}
```

Намиране на минимален и максимален елемент в масив

- **При търсене на минимален или максимален елемент трябва да се обходят всички елементи на масива**, ако той не е сортиран**.** Ако масивът е сортиран взимаме първия или последния елемент (зависи какво търсим и в какъв ред е сортиран масива).
- **Логиката на алгоритмите за намиране на минимален елемент е следната:** в променлива за минимален елемент се записва стойността на първия елемент на масива. В цикъл се обхождат всички останали елементи. При всяка итерация, текущият елемент се сравнява с минималния. Ако текущият елемент е по-малък от минималния, то той става минимален, като се присвоява на променливата за минимален елемент. При приключване на цикъла, променливата за минимален елемент съдържа търсената стойност.

Намиране на минимален и максимален елемент в масив - решение

```
int min(int arr[], int size) {
    int min = arr[0];                   // първия ел. е минимален
    for (int i = 0; i < size; i++) {    // при всяка итерация
        if (arr[i]<min) {               // ако текущия е по-малък от минималния
                min = arr[i];           // той става минимален
        }
    }
    return min;                         // връщаме min елемент
}

int main() {
    ...
    cout << "мин: " << min(arr, size) << '\n';

    return 0;
}
```

Разликата при функцията max е в сравнението на текущия елемент от масива с текущия максимален.

Алгоритми за сортиране (1)

- Сортирането подрежда елементите на списък във възходящ (нарастващ) или низходящ (намаляващ) ред.
- В следващите примери са реализирани различни алгоритми за сортиране върху масив от числа.
- Алгоритмите могат да се прилагат върху списъци от всякакви сложни обекти – низове, обекти от класове/структури за хора, продукти.
- Условието за сортиране на списъци е обектите в тях да могат да бъдат сравнявани по една или повече техни характеристики.
- Някои от алгоритмите за сортиране се изпълняват по-бързо, ако масивът в някаква степен (случайно) е предварително подреден, а при други алгоритми броят на операциите е константен.

Алгоритми за сортиране (2)

- По-елементарни за реализация са алгоритмите **сортиране чрез вмъкване, сортиране чрез пряка селекция** и **метод на мехурчето**. При тях сложността в най-лошия случай, (когато елементите са подредени в обратен ред) и в средния случай (когато елементите са равномерно разбъркани), е $O(n^2)$.
- Алгоритмите бързо сортиране (quick sort) и сортиране чрез двоично дърво са по-добри и имат сложност O(n*log(n)). Бързото сортиране с избор на един главен елемент в най-лошия случай има сложност $O(n^2)$.

Стандартни алгоритми за сортиране

- В библиотеката algorithm, има реализирана функция за сортиране sort(<указател към начален елемент>, <указател към краен елемент>), като указаният краен елемент не участва в сортирането.
- Указател към n-тия елемент на масив arr се получаваше като сума: arr + n.
- С тази функция, очевидно, може да се сортира целия масив или само желана част от него.

Стандартна функция за сортиране - пример

```
#include <iostream>
#include <string>
#include <algorithm> // за метода sort


int main() {
    setlocale(LC_ALL, "bg");

    const int size = 7;
    int arr[size] = {5, 6, -3, 8, -2, 1, 5 };

    sort(arr + 1, arr + 5);
    cout << "Сортиран масив от 2-рия до 5-тия елемент:  " << arrayToString(arr, size) << '\n';

    sort(arr, arr + size);
    cout << "Сортиран масив: " << arrayToString(arr, size) << '\n';

    return 0;
}
```

```
Сортиран масив от 2-рия до 5-тия елемент:  {5, -3, -2, 6, 8, 1, 5}
Сортиран масив: {-3, -2, 1, 5, 5, 6, 8}
```

Сортиране чрез вмъкване - идея

Идеята на алгоритъма за сортиране чрез вмъкване е следната:

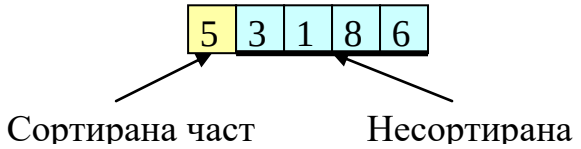
1. Списъкът за сортиране се разделя на две части: сортирана и несортирана;
2. При всяка стъпка се взема първият елемент от несортираната част и се вмъква на подходяща позиция в сортираната част. Това става като:
    a. Несортираният елемент се сравнява с всеки от елементите в сортирания списък;
    b. ако несортираният елемент е по-малък, той си разменя мястото с текущия елемент на масива, като по този начин несортираният елемент се придвижва напред;
    c. когато при сравнение несортираният елемент се окаже по-голям от текущо проверявания сортиран елемент, това означава, че той си е намерил правилната позиция.
3. Алгоритъмът приключва при изчерпване на списъка с несортираните елементи.

Така се извършва сортировка на елементите във възходящ ред. За да се сортират в низходящ ред, сравнението в точка 2.b трябва да е обратното – разменят се местата ако несортираният елемент е по-голям от текущо проверявания.

Сортиране чрез вмъкване – идеи за реализация

1. Нека разгледаме как се извършва сортировка във възходящ ред на примерен масив със стойности 5, 3, 1, 8, 6.

2. В началото (точка 1) може да приемем, че първият елемент е сортиран, а всички останали са в несортираната част.



Втора част от алгоритъма (точка 2) се изпълнява в следния цикъл:

a. Взима се i-тия елемент, чиято стойност се изменя от 1 до последния индекс (в случая 4).
b. За да се намери мястото на i-тия елемент измежду сортираните, се обхождат елементите пред него в друг вложен цикъл.
    a. Във вложения цикъл индексите, означени с j се изменят от i-тия елемент към нулевия.
    b. Сравнява се несортираният елемент (точка 2.a) с всеки j-ти сортиран елемент и ако несортираният има по-малка стойност, двата елемента си разменят местата. Във вътрешния цикъл не е необходимо да се обхождат всички сортирани елементи, а само докато се срещне елемент, който е по-малък от несортирания.
    c. Вътрешният цикъл приключва или с последната проверка (когато j стане равно на 0) или ако j-тия елемент е по-малък от несортирания елемент.

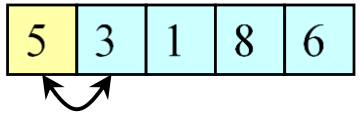
Сортиране чрез вмъкване – реализация

```
// Сортиране чрез вмъкване
void insertionSort(int arr[], int size) {

    for (int i = 1; i < size; i++) {  // обхождаме всички несортирани елементи

        int unsorted_element = arr[i]; // запомняме първия (i-ти) несортиран елемент

        int j; //ще използваме j след цикъла, затова я декларираме над него

        // във вътрешния цикъл избутваме назад с по един
        // сортираните елементи, докато са по-големи от несортирания
        for (j = i; j > 0 && unsorted_element < arr[j - 1]; j--) {
                arr[j] = arr[j - 1];
        }

        // несортираният елемент поставяме на мястото на
        // последния по-голям от него сортиран елемент
        arr[j] = unsorted_element;
    }
}
```

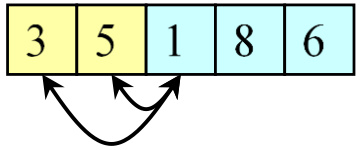
Сортиране чрез вмъкване – използване

```
int main() {
    ...
    insertionSort(arr, size);
    cout << "Сортиран масив: " << arrayToString(arr, size) << '\n';

    return 0;
}
```

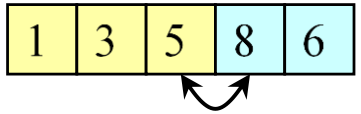
Сортиране чрез вмъкване – разяснения



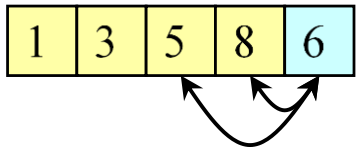
i=1, unsorted_element = 3
Във вложения цикъл се сравняват стойностите на първия и втория елемент на масива (5>3), и те разменят местата си.



i=2, unsorted_element = 1
Във вложения цикъл се сравнява 1 с елементите 5 и 3 и тъй като са по-големи, се избутват назад, а несортираният се премества в началото.



i=3, unsorted_element = 8
Във вложения цикъл се сравнява 8 с елемента 5 и не се извършва размяна, защото 8>5.



i=4, unsorted_element = 6
Във вложения цикъл се сравнява 6 с елементите 8 и 5. Разменят се само само 6 и 8. Не се проверяват други елементи, защото 6 е по-голямо от 5.

Край

Сортиране чрез вмъкване – предимства и недостатъци

- **Недостатък** на този метод е, че се извършват множество разместванията на елементи.
- **Предимство** е, че вътрешният цикъл може да приключва бързо, без да се правят всички проверки.
- В най-добрия случай **сложността на алгоритъма** може да е O(n), но обикновено тя е $O(n^2)$.

Сортиране чрез пряка селекция – идея

Логиката на сортирането чрез пряка селекция е следната:

1. Списъкът за сортиране се разделя се на две части: сортирана и несортирана.
2. Намира се най-малкия елемент в несортираната част.
3. Разменят се местата на намерения минимален с първия елемент в несортираната част.
4. Първият елемент от несортираните става последен от сортираните, а несортираната част намалява с един елемент.
5. Алгоритъмът продължава с точка 2, докато свършат несортираните елементи.

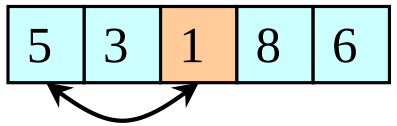
Сортиране чрез пряка селекция – идея за реализация

- За реализацията на този алгоритъм отново са необходими два цикъла:
  - **във външния цикъл** последователно **се взима i-тия елемент, като i се променя от 0 до предпоследния**;
  - **във вътрешен цикъл се намира индексът на минималния измежду елементите от i-тия до последния**. След това се **разменя намереният минимален с i-тия**.

Сортиране чрез пряка селекция – реализация

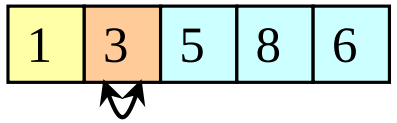
```
void selectionSort(int arr[], int size) {
    // Обхождаме всички елементи без последния.
    // Накрая последния ще си бъде на мястото.
    for (int i = 0; i < size - 1; i++) {
        // намираме индекса на минималния елемент
        // в частта от масива в интервал [i, size]
        int indexOfMin = i;
        for (int j = i; j<size; j++) {
            if (arr[j]<arr[indexOfMin]) {
                indexOfMin = j;
            }
        }

        // размяна на i-тия и минималния елемент
        int temp = arr[i];
        arr[i] = arr[indexOfMin];
        arr[indexOfMin] = temp;
    }
}
```

Сортиране чрез пряка селекция – разяснения



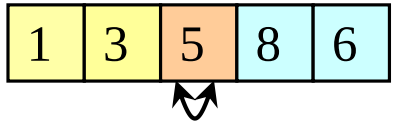
i=0

Във вложения цикъл намираме индекса на минималния елемент 1: indexOfMin→2. Разменяме го с нулевия (i=0).



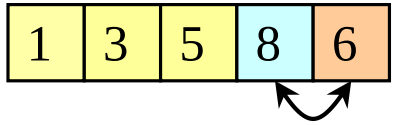
i=1

Във вложения цикъл намираме индекса на минималния елемент 3: indexOfMin→1. Разменяме го с i-тия (i=1) т.е. остава си на мястото.



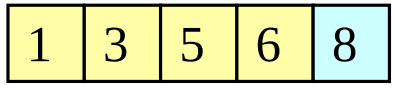
i=2

Във вложения цикъл намираме индекса на минималния елемент 5:indexOfMin→2. Разменяме го с i-тия (i=2) т.е. остава си на мястото.



i=3

Във вложения цикъл намираме индекса на минималния елемент 6: indexOfMin→4. Разменяме го с i-тия (i=3).



Край

Сортиране чрез пряка селекция – предимства и недостатъци

- **Недостатък на този метод е**, че при търсенето на минимален елемент се обхождат всички несортирани елементи. Така броят на извършваните операции винаги е еднакъв – $O(n^2)$, без значение дали списъкът е частично сортиран или не.
- **Предимството му** пред метода за сортиране чрез вмъкване е, че се извършват по-малък брой размени на елементи.

Метод на мехурчето – идея

- **Идеята на метода на мехурчето** се осъзнава най-лесно при вертикалното представяне на списъка.
- **Като се започне от края на списъка, изкачвайки се нагоре, се извършва сравняване на всеки два съседни елемента** (първо последния с предпоследния, после предпоследния с пред-предпоследния и т.н.) и **ако е необходимо, те си разменят местата така, че те да са сортирани**. ***При първото обхождане най-лекият (най-малкият) елемент (мехурчето) ще изплува най-отгоре – на позиция 0.***
- **При всяко обхождане на списъка**, **действията (сравнения и размени) се извършват само върху неподредените елементи и най-лекият от тях изплува най-отгоре**.
- **Сортирането приключва**, ако при някое от обхожданията не е направена нито една размяна.

Метод на мехурчето – реализация 1

- Първо нека разгледаме реализация, при която не се прави проверка за размяна:

```
void bubbleSort(int arr[], int size) {
    // Обхождаме масива „size-1“-пъти
    for (int i = 1; i<size; i++) {
        // Променяме j от последния до i-тия индекс със стъпка -1
        for (int j = size - 1; j >= i; j--) {
            // Ако j-тия ел. е по-малък от предходния, ги разменяме
            if (arr[j]<arr[j - 1]) {
                int temp = arr[j];
                arr[j] = arr[j - 1];
                arr[j - 1] = temp;
            }
        }
    }
}
```

Метод на мехурчето – още идеи

- **За да се проверява за размяна във външния цикъл**, се използва булева променлива, която се инициализира със стойност false.
- **Ако във вътрешния цикъл се извърши промяна**, стойността ѝ става true.
- **Ако във външния цикъл не се е извършила промяна**, изпълнението на програмата приключва.

Тази логика е добавена в следващия код. Добавен е и код за трасиране, показващ състоянието на масива при всяка итерация на вътрешния цикъл.

Метод на мехурчето – реализация 2

```
void bubbleSortTrace(int arr[], int size) {
    cout << "i,j - " << arrayToString(arr, size) << '\n';

    for (int i = 1; i<size; i++) {
        bool haveChange = false;

        for (int j = size - 1; j >= i; j--) {
            if (arr[j]<arr[j - 1]) {
                int temp = arr[j];
                arr[j] = arr[j - 1];
                arr[j - 1] = temp;
                haveChange = true;
            }
            cout << i << "," << j << " - " << arrayToString(arr, size) << '\n';
        }

        if (!haveChange) {
                break;
        }
    }
}
```

Метод на мехурчето – резултат от изпълнението

Резултат, в който показваме стойностите на i и j, и в който сме оцветили текущо сравняваните елементи, след евентуалната им размяна:

```
i,j - {5, 3, 1, 8, 6} – начален масив
1,4 - {5, 3, 1, 6, 8}
1,3 - {5, 3, 1, 6, 8}
1,2 - {5, 1, 3, 6, 8}
1,1 - {1, 5, 3, 6, 8}
2,4 - {1, 5, 3, 6, 8}
2,3 - {1, 5, 3, 6, 8}
2,2 - {1, 3, 5, 6, 8}
3,4 - {1, 3, 5, 6, 8}
3,3 - {1, 3, 5, 6, 8}
```

Двоично търсене - идея

При търсене на стойност в подреден масив, не е нужно да се обхождат всичките му елементи.

- Търсената стойност се сравнява със средния елемент от масива.
- Ако двете стойности съвпадат, методът връща индекса на текущия елемент.
- Ако търсената стойност е по-малка от средния елемент, търсенето продължава само в частта с по-малките елементи. Иначе – в частта с по-големите елементи.
- Търсенето се повтаря върху смаляващите се части от масива, докато се намери елемент или не останат елементи за претърсване.

Двоично търсене – идеи за реализация

- За организиране на търсенето, се използват три променливи за ляв, десен и среден индекс.
- Средният индекс се изчислява като средно-аритметично на левия и десния индекси.
- Ако средният елемент е по-голям търсения, търсенето продължава в лявата част на масива, като за целта десният индекс става равен на „средния индекс-1”.
- В противен случай левият индекс става равен на „средния индекс+1”.
- При всяка стъпка масивът, в който се извършва търсенето, намалява наполовина.

Двоично търсене – реализация

```
int binarySearch(int arr[], int size, int searchVal) {
    int left = 0;
    int right = size - 1; // последният валиден индекс
    while (left <= right) {
        int middle = left + (right - left) / 2;
        if (arr[middle] == searchVal) return middle;
        if (searchVal < arr[middle]) right = middle - 1;
        else                         left = middle + 1;
    }
    return -1;
}
```